**Information Systems Institute**

Distributed Systems Group (DSG)
VL Distributed Systems Technologies SS 2012 (184.260)

**Assignment 3**

Submission Deadline: 8.6.2012, 18:00

## General Remarks

- Group work is not allowed in the lab. You have to work alone. Discussions with colleagues (e.g., in the forum[1]) are allowed but the code has to be written alone.
- No deadline extensions are given. Start early and, after finishing your assignment, upload your submission as a zip file to TUWEL. If you think that you will be hard-pressed making the deadline you should upload a first version well before time runs out! We will grade whatever is there at the deadline, there is no possibility to submit later on.
- Make sure that your solution includes all the libraries and dependencies and it compiles without errors. If we cannot compile your solution you cannot get all the points.
- You should use Java 1.6.14+[2], MySQL 5.1[3] as your database management system and GlassFish v3 (you may choose either **v3-final**[4] as in assignment 2, or, if you prefer, the latest version **3.1.2-final**[5]) as your application server.
- For the AspectJ development in Task 3, we suggest to use the Springsource Toolsuite (STS)[6], although we do not require it. However, STS eases the development of aspects and advices significantly, as STS includes tooling for syntax-checking and matching of joinpoint definitions (i.e., STS tells you without running the application what your current joinpoints actually match in your application).
- For your solution, use the provided project stubs:

  All needed libraries are already included  if you want to use another one, feel free to integrate it. We expect (as can be seen in the persistence-unit configuration of persistence.xml) that you setup a database dst that can be accessed by root (without a password). You may of course change the settings of the configuration for your work at home, but please reset them to the original values in your submitted solution (and make sure it still works). Build scripts for ant[7] are already included, you may alter them, but the submitted solution has to be compilable and runnable with the predefined targets.
- The template again contains a setup task, which you should use to create the data sources necessary for the task (same as in the last assignment).

[1] https://www.infosys.tuwien.ac.at/teaching/courses/dst/forum/
[2] http://www.oracle.com/technetwork/java/javase/downloads/index.html
[3] http://dev.mysql.com/downloads/mysql/5.1.html#downloads
[4] http://glassfish.java.net/downloads/v3-final.html
[5] http://glassfish.java.net/downloads/3.1.2-final.html
[6] http://www.springsource.com/developer/sts/
[7] http://ant.apache.org/

# A. Code Part

## 1. Messaging (18 Points)

In this task you will create a simple JMS-based messaging application for the grid management system.

Please study the build file (**server/build.xml**) to understand the deployment process and complete the dist target, such that the classes the clients need are included within the client library. In the clients build files (***/build.xml**) you have to complete the specified run-*-targets we will use to start your clients with.

Another file to study is the JMS configuration file (**server/jms_config.xml**). Once you understand the concepts of JMS and Message-Driven Beans (MDB) you should be able to add all the **queues** and **topics** required for this task. An exemplary definition of one queue is already given in the file, so defining additional resources should be straightforward.

In the last assignment we have focused on assigning and processing jobs for a given number of CPUs. Now it's time to have a look at how the system processes jobs where the number of CPUs is not known in advance or was simply not specified.

After a job is assigned by a user, the grid's scheduler takes care of the scheduling and processing of the job. To do that the scheduler creates a new task (which wraps the job) and sends it to the clusters. One of the clusters takes the task, rates the complexity and decides whether one of his computers is able to process this task or not. If the task can't be processed the scheduler will be informed. In the other case (i.e., if the task can be processed), the task is forwarded to the computers. Every computer in our system belongs to one cluster and is responsible for a certain task-complexity. According to that, the task is processed by the respective computer. For example if cluster c1 rates a task as `EASY`, the task is processed by a computer that belongs to cluster c1 and is responsible for `EASY` tasks. When the computer has processed the task, the scheduler will be informed.

In reality, the steps in the procedure above can be performed mostly automatically. However, for testing purposes and to illustrate the correct functioning of your solution, you will simulate the procedure using commands that are typed manually via a command line client (see below 1.a, 1.b and 1.c).

**Your task in the following is to design and implement the described communication based on a message queuing approach**.

To keep things simple, there is only one persistent entity you need to manage in the application this time: the `Task` (Figure 1). Besides the obligatory id, this entity contains information about the job, a status, the name of the cluster, which was responsible for rating the task, as well as the complexity of the task. The status field represents the current state of the task in the process described above. The complexity field is set after the cluster in charge has rated the task.

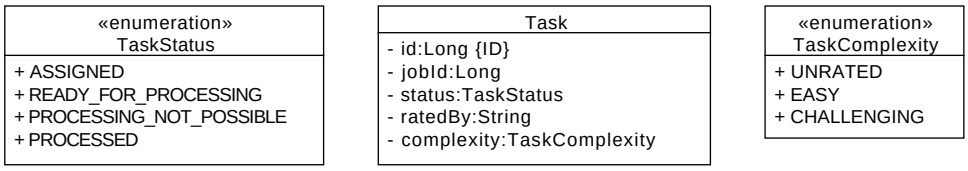


Figure 1: Task Entity

In our scenario a **server** provides the communication infrastructure. Whenever messages are exchanged (between the scheduler and the clusters or the cluster and the computers), the server is responsible for forwarding the messages. **Clients never communicate directly with each other**. This way it is also possible to keep the information about tasks up to date and to store the new status (and any other information that may have changed) to the database.

In total, the clients of our messaging system are (1) the scheduler, (2) possibly multiple clusters and (3) various computers, all of which may act as senders and receivers.

### 1.a. Scheduler

At any point in time, there is **exactly one** active scheduler that sends to and receives messages from the server. Sending and receiving messages takes place concurrently. To send messages, the application listens to user input. The scheduler provides the following console (command line) commands:

- `assign <job-Id>`

  Advises the server to create a new Task. The server creates an entry in the database and automatically forwards the task to the next available cluster. The status of the task is set to `ASSIGNED` and the complexity is set to `UNRATED`. In return to this command, the scheduler receives the *id* of the newly created task. This id is needed for the following request (it is enough to simply write the id to the console).

- `info <task-Id>`

  Advises the server to send information about a task identified by the respective id. This data must also contain all the relevant fields. It is sufficient to print the information to the console, again.

- `stop`

  Exits the application.

### 1.b. Clusters

There may be several clusters listening to the server concurrently. Each of the clusters is identified via a unique name. The server does not know which or how many clusters there are at any given moment, but you can assume there will be at least one at any time. However, it is absolutely important that every task is handled by exactly one cluster **(not more!)**. Do not implement any additional commands than the ones stated in the following for this.

Once a cluster is entrusted with a new task (which is simply printed out to the console), the cluster automatically stops listening to the server. **A cluster is never executing more than one task at any time**. The cluster provides the following commands:

- `accept <task-complexity>`

  Indicates that the cluster rated the task with either `EASY` or `CHALLENGING` and sends a message to the server. The cluster application should automatically add all required information to this message so the server can identify the respective task, update the ratedBy field (the name of the cluster), the complexity and the status to `READY_FOR_PROCESSING` in the database. After the update the server sends the task to the computers.

- `deny`

  Indicates that the cluster's computers are not able to process this task. The application sends a message to the server and automatically adds all required information to this message so the server can identify the respective task, update the ratedBy field (the name of the cluster) and the status to `PROCESSING_NOT_POSSIBLE` in the database (the complexity field shall not be updated). After the update the server informs the scheduler about the denied task. On the scheduler side, it's enough to print the information to the console.

- `stop`

  Exits the application.

### 1.c. Computers

Every Computer belongs to one Cluster and is responsible for exactly one task complexity. The server defines a special communication endpoint for all computers that are listening for tasks that were rated by their cluster and have their task complexity. Therefore, the server application can simply label the request with the cluster's name (the task's ratedBy field), the complexity of the task and propagate it to this endpoint. You can assume that there is at least one computer (possibly more) listening for a certain cluster and complexity, and that all responsible computers compute the task simultaneously and collaboratively.

The computer must be designed in a way that it only receives that messages that are intended for it (cluster and complexity) using the labels the server added to the message (check the possibilities to do this with JMS). Your infrastructure should also be able to deal with computers that are currently not listening, otherwise the request might get lost. The Computer supports the following commands:

- `processed <task-id>`

  Indicates that the computers have finished the processing of the task successfully. The server can finally update the task's status to `PROCESSED` and informs the scheduler. You can assume that the computers (which execute a task collaboratively) coordinate themselves to make sure each computer is finished before this command is sent. The server performs the status change immediately, which means that only one of the computers has to send the `processed` command. Hence, if more than one of the involved computers send this command, the first received command leads to the status change and the remaining ones have no effect (since the status is already set to `PROCESSED`).

- `stop`

  Exits the application.

You should now think about an appropriate message queuing infrastructure and decide about the features the respective queues and topics should provide to their clients. Your solution has to satisfy all the requirements stated above and should be as simple as possible. After configuring this communication infrastructure, implement the server application and all three clients.

**Never use the persistent entity for transmission directly**. Instead create DTOs yet again (like in Assignment 2), containing only the relevant information (plain text messages are not sufficient in this assignment, you should be using object messages). The server has to update the information about the persistent tasks every time it retrieves relevant messages.

In case of failures (like unknown task ids, commands to already processed tasks, ...) no distributed communication is necessary. However, your application should be able to deal with such sort of requests.

## 2. Complex Event Processing (12 Points)

In the following we take a closer look at Complex Event Processing (CEP), a technique that is becoming more and more important in today's business processes and loosely coupled distributed systems. Generally speaking, an "event" is anything (i.e., a phenomenon, happening or similar) that is of interest for the application. CEP collectively refers to techniques for processing of event messages, including event routing, event aggregation, event pattern detection, etc.

The focus of this exercise is to perform event-based queries (which continuously process the new incoming events) and identify event patterns related to the processing of Grid computing tasks in our scenario. For this exercise, we slightly extend the processing flow of the messaging example, and assume that each `Task` assigned to the grid can traverse back and forth between states during its lifetime. In particular, we assume that a `Task` changes from state `ASSIGNED` to `READY_FOR_PROCESSING`, and then either to state `PROCESSED` (if everything goes well), or on to state `PROCESSING_NOT_POSSIBLE` and back to `READY_FOR_PROCESSING` (if an error occurs during processing). Overall, we are interested in the transition between states `ASSIGNED` and `PROCESSED`, and in the transition(s) between states `READY_FOR_PROCESSING` and `PROCESSING_NOT_POSSIBLE`.

Your implementation will be based on Esper[8], a popular open-source engine for CEP. Esper provides the Event Processing Language (EPL), a powerful SQL-like language which covers numerous features tailored to efficient processing of queries over event streams. The detailed Esper reference can be found here[9]. A good starting point for your own work is the Esper tutorial[10]. The required JARs are already included in the template. Put your code into the `2_eventing` project and use the ant target named **run-eventing** to test your implementation. This time, you do not need to store any entities to the database; simply keep the `Task` instances in memory using Esper. For simplicity, the following description speaks of `Task` instances, but in fact you should again use the Task DTOs (i.e., do not feed the entity classes into Esper).

The detailed requirements are as follows:

- Initialize the Esper platform and obtain an instance of `EPServiceProvider` and `EPAdministrator`. Register the `Task` class from exercise 1 of this assignment as an event type with Esper. Provide a method to pass `Task` objects as events to Esper. You may choose an arbitrary name for the `EPServiceProvider` instance, e.g., "EsperEngineDST".
- Use the dynamic type definition capabilities of Esper (hint: `"create schema ..."`) to define 3 event types that can be derived from our `Task` event type. The names of the event types are `TaskAssigned`, `TaskProcessed` and `TaskDuration`. Each of the three types should have a `jobId` property (which corresponds to `jobId` in the `Task` class). `TaskAssigned` and `TaskProcessed` have a `timestamp` property, and `TaskDuration` has a `duration` property. All of the mentioned properties are of type `long`. Note that you should register these event types in Esper *without* creating any corresponding Java classes.
- Create and execute 3 EPL queries which generate events for the three types `TaskAssigned`, `TaskProcessed` and `TaskDuration`. Evidently, `TaskAssigned` is triggered when a `Task` event with status `ASSIGNED` occurs, and `TaskProcessed` is triggered in case of a `Task` event with status `PROCESSED`. The `TaskDuration` events combine the information of the two previous task types and contain the processing duration. Be sure to correlate the `jobId` property of the `TaskAssigned` and `TaskProcessed` types in your query. You can assume that the `TaskDuration` query does not need to consider more than 10.000 historical events (i.e., there are never more than 10.000 events between any two correlated Task events).
- Implement the following three EPL queries and provide a listener which outputs (to stdout) new events that result from executing the queries:
  - Receive notifications about all events of type `TaskDuration`.
  - Upon arrival of each `TaskDuration` event, emit an event that computes the average execution duration over all tasks that finished within the last minute.
  - Use pattern matching facilities of EPL to detect tasks which have 3 times attempted and failed to execute (i.e., switched 3 times between the status `READY_FOR_PROCESSING` and the status `PROCESSING_NOT_POSSIBLE`).
- Provide a reasonable test sequence of input events (type `Task`) which covers at least one result for each of the implemented queries. Your test data should comprise at least 10-20 `Task` instances with a non-trivial lifecycle. Moreover, the lifecycles of the tasks should not be strictly chronological, but the events for different tasks should overlap (e.g., `task1.ASSIGNED` - `task2.ASSIGNED` - `task1.READY_FOR_PROCESSING` - `task3.ASSIGNED` - ...). If you choose to simulate a random sleep period between the individual events, please ensure that test finishes within reasonable time (say, 30 seconds). Your test class should create the Task instances and feed them directly to the Esper querier, i.e., for this task you do not need to use the messaging infrastructure from earlier to transmit and modify the status of the Tasks. Be sure to thoroughly test your solution - event-based and asynchronous processing is inherently prone to timing and synchronization faults. A sample output of your test program could look like this:

```
[java] Duration: 900ms
[java] Avg duration: 900.0ms
```

[8] http://esper.codehaus.org/
[9] http://esper.codehaus.org/esper-4.5.0/doc/reference/en/html_single/
[10] http://esper.codehaus.org/tutorials/tutorial/tutorial.html

```
[java] Duration: 200ms
[java] Avg duration: 550.0ms
[java] Duration: 700ms
[java] Avg duration: 600.0ms
[java] Duration: 200ms
[java] Avg duration: 500.0ms
[java] Detected pattern with 3 loops of (READY_FOR_PROCESSING -> PROCESSING_NOT_POSSIBLE).
[java] Duration: 500ms
[java] Avg duration: 500.0ms
[java] Duration: 600ms
[java] Avg duration: 516.6666666666666ms
[java] ...
```

Note that the example queries above cover only a very small subset of Esper's capabilities. Use the Esper reference to get familiar with some of the additional core concepts (e.g., *contexts*). During the interview sessions you should be prepared to report on your experiences with CEP and the strengths/weaknesses of Esper. Also, think about the core differences of the Esper processing model as opposed to querying a standard database like MySQL.

## 3. Dynamic Plugins Using AspectJ (14 Points)

Another important aspect of modern application servers is the dynamic loading and deployment of plugins or applications. This feature is also a nice demonstration for using reflection and class loading at runtime, so we will again implement a (simplified) custom solution of our own. Like the last task in Assignment 2, this particular task has nothing to do with the Grid case study. Put the code for this task into the **3_plugins** subprojects.

- **3.a. Plugin executor (4 Points)**

  Implement the `IPluginExecutor` interface. This is the main component responsible for executing plugins. It has to monitor (i.e., repeatedly list the contents of) several directories to detect whether new **.jar** files were copied to these directories or existing .jar files were modified. The executor then scans the file and looks for classes that implement the `IPluginExecutable` interface. If some plugin executable is found, the executor spawns a new thread and calls its execute method. For this, you should be using a thread pool. Take care of class loading: there must not be any problem with the concurrent execution of different plugins containing classes with equal names. Also make sure to free all acquired resources after the execution of a plugin has been completed. The second method in the interface (`IPluginExecutor.interrupted()`) will be important later on, you can leave this method body empty for now.

  Put your code into the **loader** directory. Complete the **run** target (in your main method you should start scanning for plugins in the **plugins** directory and stop when enter is hit).

- **3.b. Logging Plugin Executions (6 Points)**

  Now we want to implement some (decoupled) logging facilities for our plugin executor framework. To this end, we will make use of AspectJ and Aspect-Oriented Programming (AOP). The required AspectJ libraries are already part of the template. Similarly, most of the necessary configuration has already been done for you:

  - Look at the configuration file `META-INF/aop.xml`. Register all aspects you are going to write for the next two tasks here. For debugging reasons, you may want to enable the verbose mode during development (use the commented out weaver line), but please disable the verbose mode before submitting.
  - The template has been configured to use load-time weaving (i.e., the aspects are weaved into your classes at class loading time). To this end, we again use the Java agent mechanism. See the `run` target in `3_plugins/loader/build.xml`, to learn how we enabled load-time weaving.

  Before starting to develop, you should familiarize yourself with the concepts of aspects, advices, joinpoints and pointcuts (you can also tackle Theory Question 5 as you go along). Then, refer to the AspectJ Development Kit Developer's Notebook[11] for support on how these concepts are

[11] http://www.eclipse.org/aspectj/doc/released/adk15notebook/

implemented in AspectJ. Very likely, you will want to use the annotation-based development style[12] (although you are also free to implement advices the old-fashioned, stand-alone way, if you feel like it).

Your first task is now to write a simple logging aspect for plugins. Essentially, the aspect should write a single line of logging output before a plugin starts to execute, and after a plugin is finished. The log message can be very brief, but needs to contain the actual class name of the plugin:

```
[java] Plugin dst3.dynload.sample.PluginExecutable started to execute
[java] Plugin dst3.dynload.sample.PluginExecutable is finished
```

In some cases, users of the plugin framework might want to disable logging for some plugins. Implement a method annotation `dst3.dynload.logging.Invisible`. Whenever an `IPluginExecutable.execute()` is annotated as invisible, its execution should not be logged. Make sure that this condition is already considered in the pointcut definition of your logging advice (i.e., you should **not** match just any plugin method and filter out invisible plugins in your Java code).

Additionally, your logging aspect should re-use the logger of the plugin, if the plugin has defined one. That is, if the plugin has a member field of a subclass of `java.util.logging.Logger`, your log statements should be written to that logger. If no such logger is defined, use `System.out`. We have already configured the template to print all log messages of level INFO and higher (see `META-INF/logging.properties`).

- **3.c. Plugin Performance Management (4 Points)**

  Plugin frameworks like the one we are implementing often need some way to influence the execution of the managed plugins. Hence, we now implement some means to interrupt plugins whose execution takes too long.

  Start by creating a new method annotation (`Timeout`), which has one Long parameter. Users can use this annotation, to define the "normal" maximum execution time of their plugin, i.e., the `Timeout` should only be used to annotate `IPluginExecutable.execute()` methods. Then, create a new aspect that enforces this defined maximum execution time. If a plugin is detected that takes longer than its maximum defined time, call this plugin's `interrupted()` method. You do not need to take any further action (i.e., we can assume that the developer of the plugin actually terminates the plugin if this callback is invoked). However, keep in mind that the `interrupted()` method can itself take some time to execute, so do not block while the client is cleaning up. If no timeout is defined for a plugin, you can assume that the plugin can run for as long as it needs to.

You have to provide a **sample application** in which you implement at least 2 `IPluginExecutables` that need some time to finish. If you implemented the performance management part, also build a small example that loops endlessly until it is interrupted (at which time it should gracefully shut down). By using the **dist** target a jar file is automatically built and copied to the **loader/plugins** directory.

[12] http://www.eclipse.org/aspectj/doc/released/adk15notebook/ataspectj.html

# B. Theory Part

The following questions will be discussed during the practice lesson. At the beginning of the each lesson we hand out a list where you can specify which questions you have prepared and are willing to present. We will then select students at random who checked a question to discuss the question (you know the procedure from your math courses). If you are asked to discuss a question but fail to provide a correct and well-founded answer, you will lose **all** points for the theory part of this assignment.

## 4. Class loading (1 point)

Explain the concept of class loading in Java. What different types of class loaders do exist and how do they relate to each other? How is a class identified in this process? What are the reasons for developers to write their own class loaders?

## 5. AOP Fundamentals (2 points)

Explain the concept of Aspect Oriented Programming (AOP). Think of typical usage scenarios. What are aspects, concerns, pointcuts and joinpoints, and how do these concepts relate to each other? Why is it so important to write minimally matching pointcut definitions?

## 6. Weaving Times in AspectJ (1 point)

What happens during weaving in AOP? At what times can weaving happen in AspectJ? Think about advantages and disadvantages of different weaving times.

## 7. Esper Processing Model (2 point)

Study the details of the Esper processing model (available in the online reference of Esper). Describe the core API elements, and illustrate the main EPL query types based on an exemplary event timeline.